Vorlesung "Softwaretechnik"
Buchkapitel 11
Analytische Qualitätssicherung 2

Lutz Prechelt

Freie Universität Berlin, Institut für Informatik

- Testautomatisierung
  - Aufnahme/Wiederg.-Werkzeuge
  - Testframeworks
- Sonstiges über Testen
  - Heuristiken zur Defektortung
  - Test-Ende
  - Leistungs-, Benutzbarkeits-, Abnahmetests
- Manuelle statische Prüfung
  - Durchsichten und Inspektionen
  - Perspektiven-basiertes Lesen
  - Empirische Ergebnisse
- Automatische statische Prüfung
  - Modellprüfung
  - Quelltextanalyse

Wo sind wir?: Taxonomie "Die Welt der Softwaretechnik"

Freie Universität Berlin

Welt der Problemstellungen:

- Produkt (Komplexitätsprob.)
  - Anforderungen (Problemraum)
  - **Entwurf (Lösungsraum)**
- Prozess (psycho-soziale P.)
  - Kognitive Beschränkungen
  - Mängel der Urteilskraft
  - Kommunikation, Koordination
  - Gruppendynamik
  - Verborgene Ziele
  - **Fehler**

Welt der Lösungsansätze:

- Technische Ansätze ("hart")
  - Abstraktion
  - Wiederverwendung
  - Automatisierung
- Methodische Ansätze ("weich")
  - Anforderungsermittlung
  - Entwurf
  - **Qualitätssicherung**
  - Projektmanagement

Wo sind wir?: Qualitätssicherung

- Einsicht: Man macht beim Bau von SW zahlreiche Fehler
  - die häufig zu schwerwiegenden Mängeln führen

- Prinzipien:
  - **Konstruktive Qualitätssicherung**: Ergreife vorbeugende Maßnahmen, um zu *vermeiden*, dass etwas falsch gemacht wird (Qualitätsmanagement, Prozessmanagement)
  - ***Analytische Qualitätssicherung***: Verwende prüfende Maßnahmen, die entstandene Mängel aufdecken
  - ***Softwaretest***: dynamische Prüfung
  - ***Durchsichten***: manuelle statische Prüfung

Übersicht

Analytische QS:

- **Dynamische Verfahren**
  - **Defekttest**
    - Wie wählt man Zustände und Eingaben aus?
    - Wer wählt Zustände und Eingaben aus?
    - Wie wählt man Testgegenstände aus?
    - Wie ermittelt man das erwartete Verhalten?
    - Wann wiederholt man Tests?
    - **Wann / wie kann und sollte man Tests automatisieren?**
  - Benutzbarkeitstest
  - Lasttest
  - Akzeptanztest
- Statische Verfahren
  - ...

Konstruktive QS:

- Test- und Durchsichtsmgmt.
- Prozessmanagement
- Projektmanagement, Risikomanagement

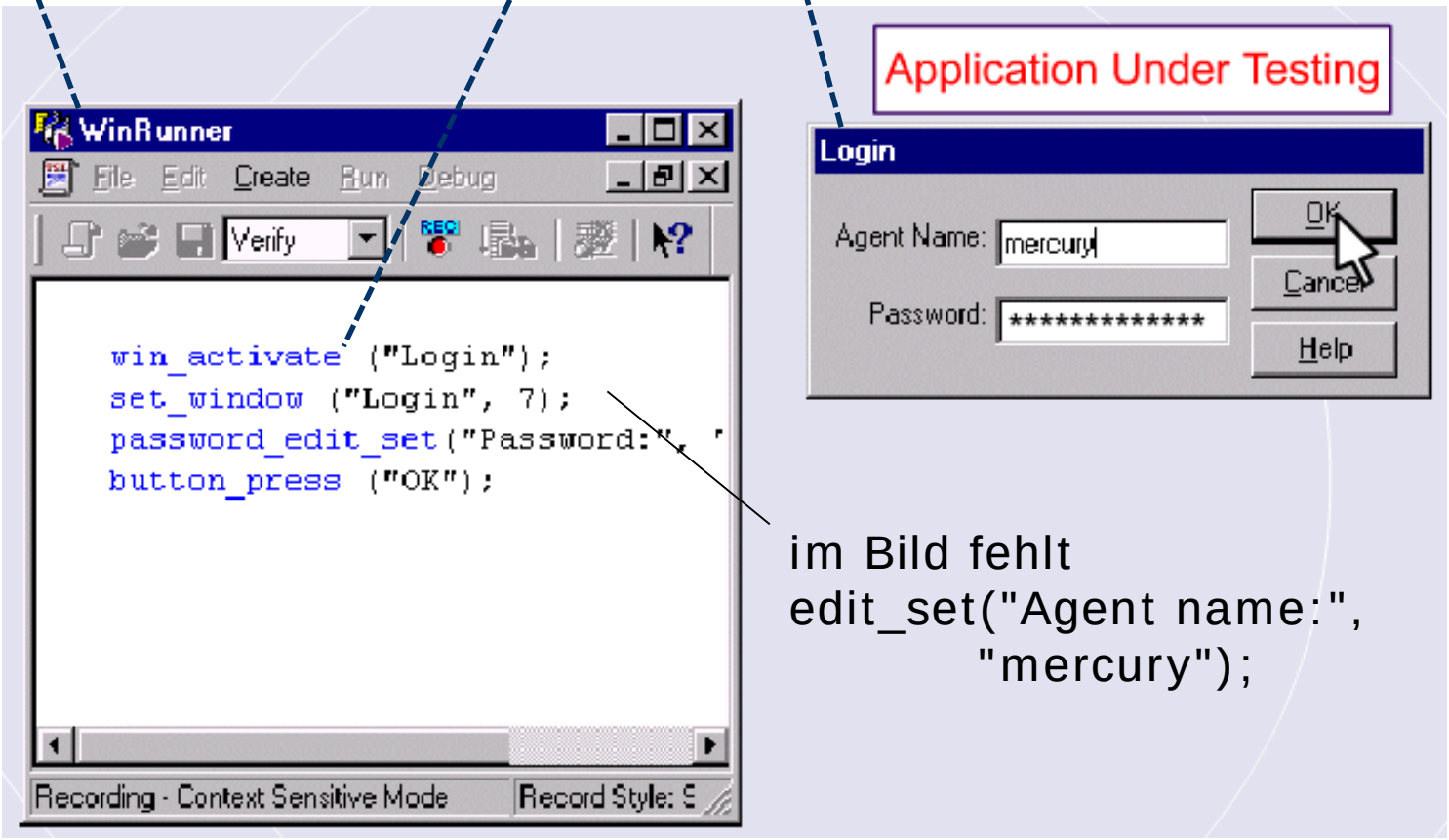
Testwerkzeuge

- Aufnahme/Wiedergabe-Werkzeuge (auch genannt: Record/Playback, Capture/Replay etc.)
  - zeichnen Bedienaktionen auf und lassen sie später wieder ablaufen
  - vor allem für GUI-Programme
- Lasttest-Werkzeuge
- Steuerungs- und Protokollprogramme
  - rufen Testtreiber auf, protokollieren die Ergebnisse, erstellen Übersichten, erzeugen Benachrichtigungen
- u.a.m. (Profiler, Speicherprüfer, ...)
- Wichtigste Hersteller:
  - HP Enterprise Mercury testing tools
    IBM Rational testing tools
    Borland Silk testing tools
  - Open Source Selenium (Browsertests)
    www.opensourcetesting.org

Aufnahme/Wiedergabe-Werkzeuge

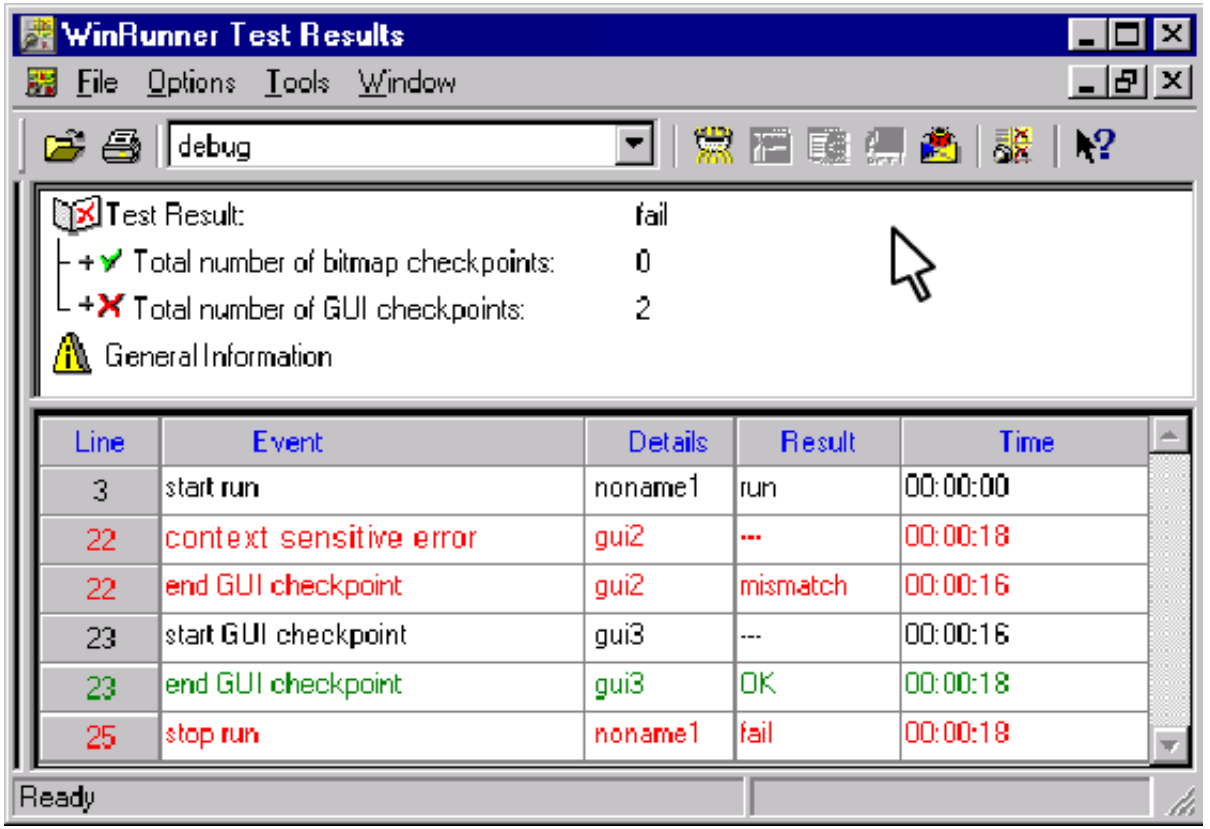
Beispiel: WinRunner von HPE Mercury

- Zeichnet auf, was am zu testenden GUI getan wird
- und speichert es als WinRunner-Skript ab



Aufnahme/Wiedergabe-Werkzeuge (2)

- Dieses Skript wird dann manuell geändert:
  - Ersatz fester Werte durch Variablen
    - Parametrisierung, Unterprogrammbildung
  - Einfügen von Resultatprüfungen
    - WinRunner kann Werte aus dem GUI ablesen:
      - Zustände von Checkboxen etc.
      - Text
      - Pixelbilder

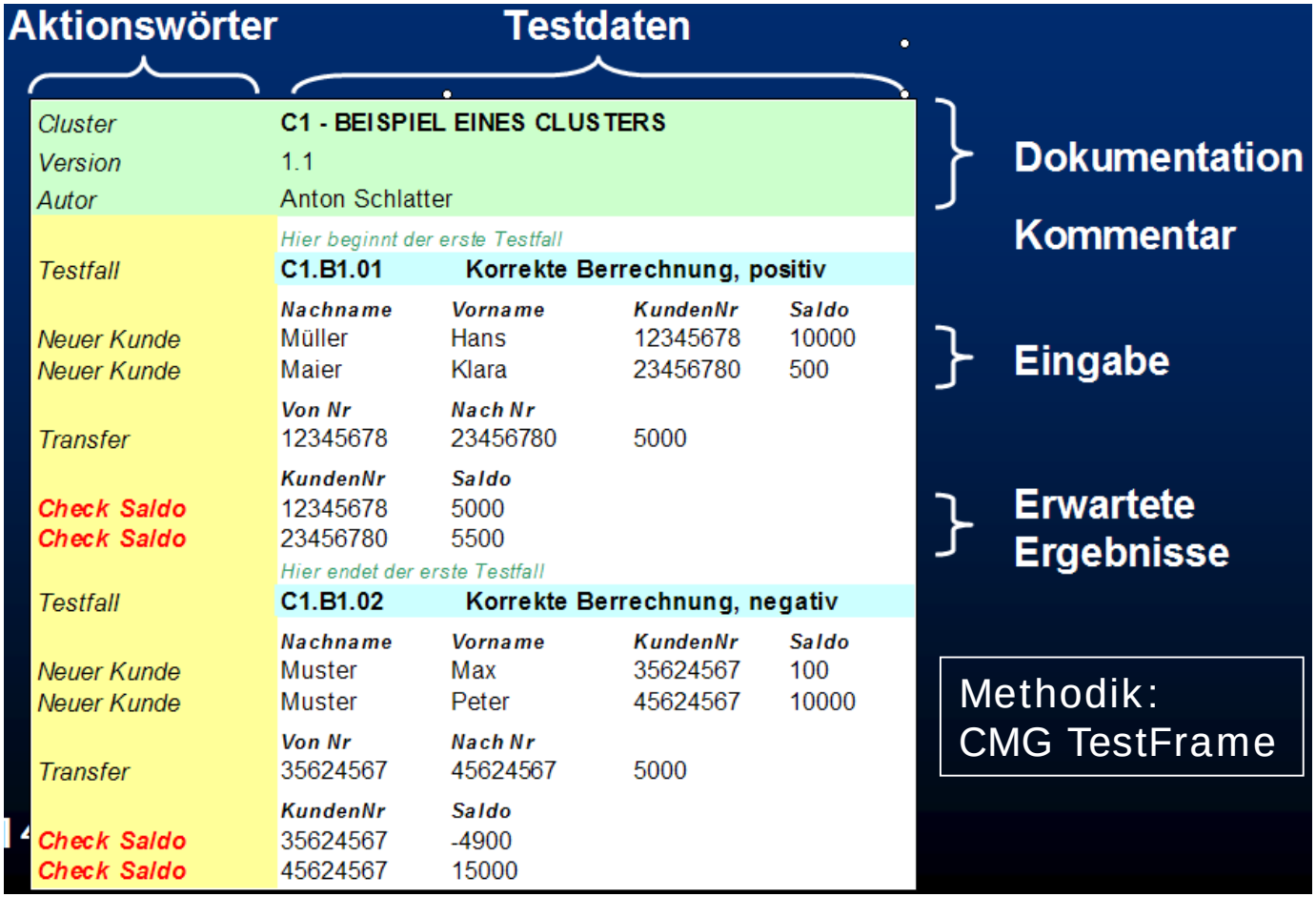


- (Es gibt andere, vergleichbare Werkzeuge)
  - Die Prinzipien sind immer ähnlich
  - auch f. Mobilgeräte

Modularisierung automatischer Tests

- Auf Basis solcher gekapselter Automatisierungsroutinen (*"Aktionswörter"*, technische Ebene) formuliert man nun die eigentlichen Testfälle (fachliche Ebene)

- Treiber-programm vermittelt zwischen generischer Notation u. einzelnem Werkzeug



Modularisierung automatischer Tests: Vorteile

Freie Universität Berlin

- Die Details des GUIs ändern sich häufiger als die Logik der Testfälle
  - Änderungen sind in Aktionswörtern gekapselt
    - ➔ Modularisierung mit Geheimnisprinzip
- Große Systeme benutzen mehrere verschiedene Automatisierungswerkzeuge
  - Testfallformulierung dennoch einheitlich möglich
  - Dadurch auch bessere Dokumentation der Testfälle
- Klare Trennung der Aufgaben:
  - Testspezialist/in (fachlich)
  - Testwerkzeugspezialist/in (technisch)
- Wechsel des Testwerkzeugs bleibt machbar

Testframework-Ansatz

Testframework =
Anwendungsspezifische Bibliothek von Test-Hilfsoperationen

- Erleichtert stark das Schreiben der Testtreiber
- Ermöglicht sauber entworfene und änderungsfreundliche Testsuites
  - für GUIs (mit Aufnahme/Wiedergabe-Werkzeug)
    - z.B. Selenium, Watir
  - für programmiersprachliche Tests
- Problem:
  - Hohe Vorab-Investition
- Mögliche Rahmen sind z.B. FitNesse, Cucumber/Gherkin

Ein generisches Testframework: JUnit

Freie Universität Berlin

- Generisches Testframework für Java
  - www.junit.org Autoren: Erich Gamma, Kent Beck
  - Nur Verwaltungsrahmen, keine anwendungsspezifische Logik
  - Sehr populär bei Firmen und im Open-Source-Bereich
    - Es gibt sehr ähnliche Werkzeuge für viele Sprachen:
    - https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

- Grundideen:
  - Testfälle liegen in einer Klasse mit annotierten Methoden:
    - @Before Zustand vor Testbeginn herstellen
    - @Test Test durchführen, Prüfungen mit *assertEquals* etc.
    - @After Zustand (z.B. in Datenbank) wieder "aufräumen"
  - Durch diese Vorgaben werden automatisierte Tests gleichmäßig und übersichtlich strukturiert
  - Das Framework ruft solche Tests auf, fängt Ausnahmen, zählt Versagen, trägt Ergebnisse zusammen

A (trivial) class to be tested

```
public class Calculator {
    private static int result;

    public void add(int n) {
      result = result + n;
    }
    public void substract(int n) {
      result = result - 1; //Defect!
    }
    public void multiply(int n) {
      // Not implemented yet!
    }
    public void divide(int n) {
      result = result / n;
    }
    public void square(int n) {
      result = n * n;
    }
    public void squareRoot(int n) {
      for (;;) ; //infinite loop: Defect!
    }
    public void clear() {
      result = 0;
    }
    public void switchOn() {
      result = 0; // etc.
    }
    public void switchOff() { ... }

    public int getResult() {
      return result;
    }
}
```

Source: http://www.devx.com/Java/Article/31983/0/

A JUnit test class

```
public class CalculatorTest
        extends TestCase{
    private static Calculator
      calculator = new Calculator();

    @Before
    public void clearCalculatr() {
      calculator.clear();
    }
    @Test
    public void add() {
      calculator.add(1);
      calculator.add(3);
      assertEquals(
        calculator.getResult(), 4);
    }
    @Test
    public void subtract() {
    ...
    }
    @Test
    public void divide() {
    ...
    }
    @Test(expected =
    ArithmeticException.class)
    public void divideByZero() {
      calculator.divide(0);
    }
    @Ignore("not ready yet")
    @Test
    public void multiply() {
      calculator.add(8);
      calculator.multiply(10);
      assertEquals(
        calculator.getResult(), 80);
    }
}
```

Execution by JUnit

When executing a test class,

FOR each method annotated **@Test**:

- JUnit calls the method annotated **@Before**
  to set up the class-specific testing environment
- JUnit calls the method annotated **@Test**
  to perform the individual tests
- JUnit catches any exceptions that are thrown
  - those thrown by '*assert*' indicate explicitly detected failures
  - others indicate other failures
    - except that those announced by *expected=SomeException.class* etc. *must* happen (and will then be ignored)
- JUnit calls the method annotated **@After**
  to deconstruct the class-specific testing environment

END FOR

- JUnit reports the number and nature of problems found

JUnit example output

- *java –ea org.junit.runner.JUnitCore CalculatorTest*

```
JUnit version 4.1
...E.EI
There were 2 failures:
1) subtract(CalculatorTest)
   java.lang.AssertionError: expected:<9> but was:<8>
   at org.junit.Assert.fail(Assert.java:69)
2) squareRoot(CalculatorTest) java.lang.Exception:
   test timed out after 1000 milliseconds at
   TestMethodRunner.runWithTimeout

FAILURES!!! Tests run: 4, Failures: 2
```

- Note: If you use 'assert' rather than only 'assertEquals' etc., bin/java must be called with option -ea to enable assertions!

JUnit: Extensions

- **@BeforeClass**, **@AfterClass**
  - static methods; run only once for all @Test together
- Table-driven tests:
  - Annotate test class with **@RunWith(Parameterized.class)**
  - Include a method **@Parameters**
    public static Collection data()
    that returns the test data table
  - The methods **@Test** have an entry of the Collection as parameter and will be called once for each of them
- Compose test classes into test suites:
  @Suite.SuiteClasses({A.class, B.class})
- Test parallelization
- Test selection and ordering heuristics
  - New tests, fast tests, and recently failing tests are executed first
- IDE integration:
  - An Eclipse plugin provides nicer call, results presentation, etc.

Defektortung (Systemebene): 9 Heuristiken

Freie Universität Berlin

1. Verstehe das System
   - (auch wenn's schwerfällt)
2. Reproduziere das Versagen
   - (auch wenn's schwerfällt)
3. Nicht denken, hingucken!
   - selbst wenn Du glaubst Du weisst was los ist
4. Teile und herrsche
   - Keine voreiligen Schlüsse bitte
5. Ändere immer nur eine Sache
   - selbst wenn sie trivial erscheint
6. Mach Notizen was passiert
   - schriftlich!
7. Prüfe Selbstverständlichkeiten
   - zumindest nach einiger Zeit vergeblicher Suche
8. Frag Außenseiter um Rat
   - zumindest nach einiger Zeit vergeblicher Suche
9. Wenn Du es nicht repariert hast, ist es auch nicht repariert
   - also repariere es und prüfe dann nochmal nach

Mehr dazu im **Kurs "Debugging"** (zuletzt im SoSe 2007)

Wann kann/sollte man mit dem Testen aufhören?

Freie Universität Berlin

- Im Prinzip: Wenn die Kosten zum Aufdecken weiterer Defekte den Nutzen, sie entdeckt zu haben, übersteigen
- Konkret kennt man aber weder die Kosten noch den Nutzen
  - es gibt aber zumindest Schätzmodelle für die Defektanzahl

Typische Lösungen in der Praxis (auf Systemebene):

- Häufig: Test endet, wenn der Zeitplan erschöpft ist
  - bzw. wenn weitere Überziehung nicht mehr akzeptiert wird
- Manchmal: Test endet, wenn neue Versagen "selten" werden
  - Sinnvoll, wenn ein kompetentes Team testet
- Manchmal: Test endet, wenn die SW laut Guru "gut genug" ist
  - d.h. ein Experte entscheidet, ohne begründen zu müssen
- Manchmal: Test endet, wenn alle Testfälle des Testplans bestanden werden (modernes Stichwort: ATDD)
  - Testplan wird in der Entwurfsphase gemacht und später ergänzt

Übersicht

Analytische QS:

- **Dynamische Verfahren**
  - Defekttest
    - Wie wählt man Zustände und Eingaben aus?
    - Wer wählt Zustände und Eingaben aus?
    - Wie wählt man Testgegenstände aus?
    - Wie ermittelt man das erwartete Verhalten?
    - Wann wiederholt man Tests?
    - Wann/wie kann und sollte man Tests automatisieren?
  - **Benutzbarkeitstest**
  - **Lasttest**
  - **Akzeptanztest**
- Statische Verfahren
  - ...

Konstruktive QS:

- Test- und Durchsichtsmgmt.
- Prozessmanagement
- Projektmanagement, Risikomanagement

Benutzbarkeitstest

- Teil des *Usability Engineering* (Benutzbarkeits-Gestaltung)
- Prüft, ob echte Benutzer/innen in der Lage sind, die Funktionen des Systems zu nutzen
  - und wo dabei Schwierigkeiten auftreten
- Verfahren ist meist die Beobachtung solcher Benutzer/innen
  - bei freier Benutzung oder
  - beim Lösen vorgegebener Aufgaben
- anschließende Verbesserung der Software
- und erneute Beobachtung

- Usability Engineering ist sehr wichtig und wertvoll
  - wird aber dennoch vielerorts immer noch nicht betrieben

Qualitätsmerkmale von Software (ganz grob)

Freie Universität Berlin

Externe Qualitätseigenschaften (aus Benutzersicht)

- Benutzbarkeit
  - Bedienbarkeit, Erlernbarkeit, Robustheit, ...
- Verlässlichkeit
  - Zuverlässigkeit, Verfügbarkeit, Sicherheit, Schutz
- Brauchbarkeit
  - Angemessenheit, Geschwindigkeit, Skalierbarkeit, Pflege, ...
- ...

(Man kann diese Listen auch ganz anders machen.)

Ziel von Benutzbarkeitstests

Ziel von Last- / Stresstests

Interne Qualitätseigenschaften

- Zuverlässigkeit
  - Korrektheit, Robustheit, Verfügbarkeit, ...
- Wartbarkeit
  - Verstehbarkeit, Änderbarkeit, Testbarkeit, Korrektheit, Robustheit
  - dafür nötig sind z.B. Dokumentiertheit, Flexibilität, etc.
- Effizienz
  - Speichereffizienz, Laufzeiteffizienz, Skalierbarkeit
- ...

Interne Q. sind Mittel zum Zweck!

Lasttest, Leistungstest, Stresstest

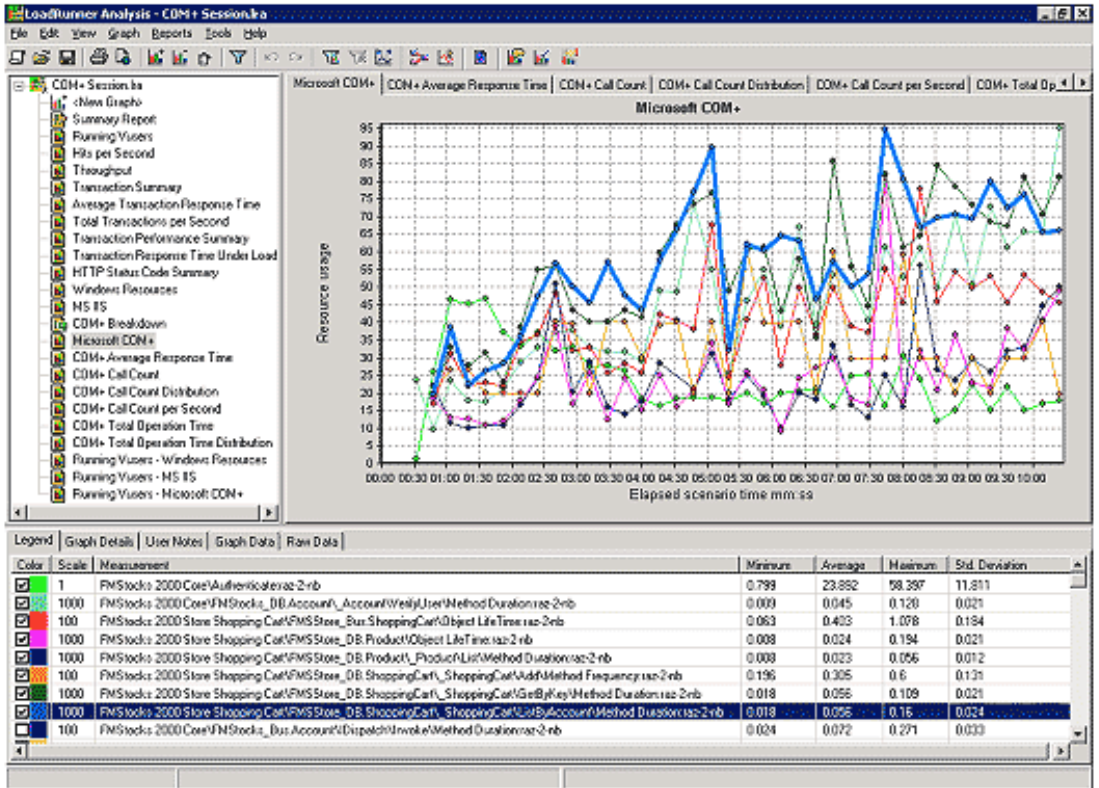
Schickt sehr "strapaziöse" Eingaben an das System

Fragen:

- **Leistungstest**: Hält das System die nötigen Antwortgeschwindigkeiten ein?
  - Wichtig bei zeitkritischen Systemen / anspruchsvollen Nutzern
- **Lasttest**: Kann das System genügend viele Benutzer/innen zugleich schnell genug bedienen?
  - Wichtig bei Mehrbenutzersystemen, insbes. im Web
- **Stresstest**: Überlebt das System auch Überlasten, massenhaft unsinnige Eingaben und ähnliches?
  - Wichtig bei lebens- und geschäftskritischen Systemen
  - Deckt sogar Defekte in Betriebssystemkernen auf („fuzzing“), siehe "Month of the Kernel Bugs" (Nov. 2006) http://projects.info-pull.com/mokb/

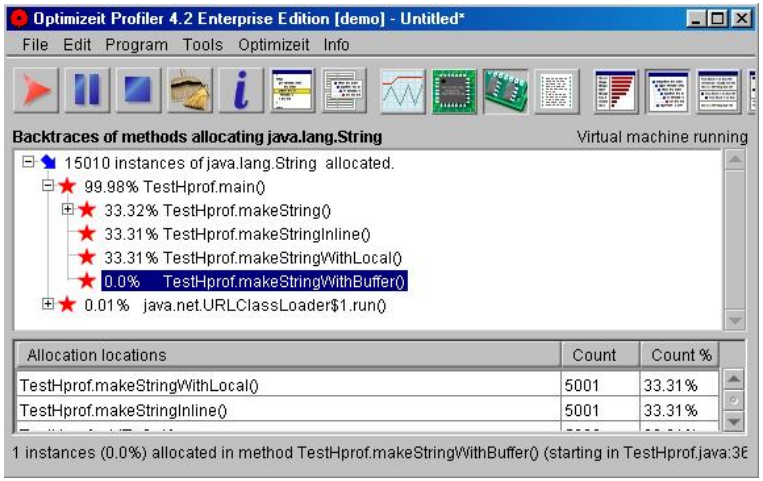
Werkzeuge für Lasttests

- Simulieren die Aktionen zahlreicher Benutzer/Aufrufer
- Werkzeuge verfügbar
  - Kommerziell: Rational Performance Tester, LoadRunner, SilkRunner
  - Open Source: JMeter, OpenSTA, The Grinder
- Für unterschiedl. Plattformen, vor allem für verteilte Systeme
  - http/https
  - RDBMs
  - EJB
  - SAP R/3
  - CORBA, COM+
  - etc.



Werkzeuge für Leistungstest

- Leistungstests dienen insbesondere als Grundlage für die Optimierung von Entwürfen und Implementierungen
  - Laufzeitverhalten; Speicherbedarf
- Hilfswerkzeuge dabei sind sogenannte Profilierer (profiler)
  - z.B. perf/hperf; http://java-source.net/open-source/profilers
- Sie erzeugen z.B. Ausgaben folgender Art:
  - Laufzeitprofilierer: *"Die Methode A wurde 126 745 mal aufgerufen, verbrauchte netto 22,6% der Laufzeit und brutto (also mit Unteraufrufen) 71,2% der Laufzeit."*
    - und das für alle Methoden
    - sowie Informationen über Threads
  - Speicherprofilierer: *"Von Klasse B wurden 1.793.233 Exemplare erzeugt (700.482 KB). Davon existieren noch 287 Exemplare (115 KB)."*



Akzeptanztest

- Dient dazu, dem Kunden zu demonstrieren, dass das Produkt nun tauglich ist

- Beachte den Wechsel des Ziels:
  - Defekttests und Benutzbarkeitstests sind erfolgreich, wenn sie Mängel aufdecken
    - denn das ist ihr Zweck
  - Akzeptanztests sind hingegen erfolgreich, wenn Sie keine (oder nur geringe) Mängel aufdecken

- Akzeptanztests sollten direkt aus den Anforderungsdokumenten (meist Use Cases) hergeleitet werden
  - Ein kluger Auftraggeber macht das selbst (oder überträgt es Dritten) und legt die Testfälle erst spät offen, jedenfalls ein paar

Gemeinsame Schwäche aller Testverfahren

Freie Universität Berlin

- Alle testenden Verfahren führen zunächst nur zu Versagen
- Das Versagen muss dann auf einen Mangel zurückgeführt werden (bei Defekttest genannt **Debugging**)
  - Siehe oben: Heuristiken für die Defektlokalisierung
- Das kann sehr **aufwändig** sein:
  1. Das in Frage kommende Codevolumen ist evtl. sehr groß
  2. Evtl. spielen mehrere Mängel zusammen
  3. Oft wirkt ein Mangel zeitlich lange bevor man das Versagen sieht

- In dieser Hinsicht sind **statische und konstruktive Verfahren günstiger**:
  - Die Aufdeckung des Mangels geschieht hier meist direkt am Mangel (insbesondere: Defekt)
  - Die Lokalisierungsphase entfällt deshalb

Übersicht

Analytische QS:

- Dynamische Verfahren (Test)
  - Defekttest
  - Benutzbarkeitstest
  - Lasttest
  - Akzeptanztest

- **Statische Verfahren**
  - **Manuelle Verfahren**
    - **Durchsichten, Inspektionen**
  - Automatische Verfahren
    - Modellprüfung
    - Quelltextanalyse

Konstruktive QS:

- Test- und Durchsichtsmgmt.
- Prozessmanagement
- Projektmanagement, Risikomanagement

Manuelle statische Verfahren

- Bei manuellen statischen Verfahren werden SW-Artefakte von Menschen **gelesen** mit dem Ziel Mängel aufzudecken

1. Arbeitsergebnisse werden gründlich auf Mängel hin durchgesehen (meist separat von 2 Personen)
   - Mängel können sein:
     - Defekte
     - Verletzungen von Standards
     - Verbesserungswürdige Lösungen
2. Mängel werden diskutiert
3. Mängel werden beseitigt
   - evtl. nicht alle (wg. Kosten/Nutzen-Abwägung)
4. Nach kritischen Korrekturen evtl. erneute Prüfung

Vorteile manueller statischer Verfahren

1. Im Gegensatz zum Test benötigen solche Verfahren keinen ausführbaren Code
   - sondern sind anwendbar auf **alle Arten von Dokumenten**: Anforderungen, Entwürfe, Code, Testfälle, Dokumentationen
     - und sogar auf Prozessdokumente wie Projektpläne u.ä.

2. Dadurch werden Mängel **früher aufgedeckt**, was viel Aufwand spart

3. Außerdem haben die Verfahren **Zusatznutzen** neben der Aufdeckung von Mängeln:
   - **Kommunikation**: Verbreitung von Wissen über die Artefakte (und damit verbundene Anforderungen und Entwurfsideen) im Team
   - **Ausbildung**: Wissenstransfer über Technologie und gute Strukturen/Stil

Begriffe

- Manuelle Prüfmethoden laufen unter vielen verschiedenen Bezeichnungen:
  - Durchsicht (review)
  - Kollegendurchsicht (peer review)
  - Inspektion (inspection)
  - Durchgang (walkthrough)
  - Formelle Technische Durchsicht (formal technical review, FTR)
  - und anderen
- Bei manchen Leuten haben manche oder alle dieser Begriffe eine recht genau festgelegte Bedeutung
- Leider nicht überall die gleiche

- Vorschlag: Meistens von "Durchsicht" sprechen
  - Und bei Bedarf präzisieren, was man damit meint

Beispiel 1: Kleine Codedurchsicht (halbformell)

Freie Universität Berlin

Peer Review:

- Entwickler A hat 4 zusammengehörige Klassen fertig gestellt
  - samt automatisierter Modultests ("unit tests")
- Er sendet Entwicklerin B eine Email und bittet, die 4 Klassen (insgesamt 600 Zeilen Code) zu begutachten
  - B kennt die Anforderungs- und Entwurfsüberlegungen, aus denen sich ergibt, was die Klassen leisten sollten
- B nimmt sich dafür 3 Stunden Zeit
- B meldet entdeckte Mängel per Email an A zurück:
  - 2 vergessene Funktionen
  - 2 Zweifel an Bedeutung von Anforderungen
  - 4 Fehler in Steuerlogik
  - 1 Fehler in Testfall
  - 5 übersehene Fehlerfälle
  - 4 Vorschläge zur Verbesserung der Robustheit
  - 3 Verstöße gegen Entwurfs-/Kodier-/Kommentierrichtlinien

Beispiel 2: Inspektion (hoch formell)

- Bekanntestes Verfahren
  - Vorgeschlagen von M. Fagan und H. Mills bei IBM ca. 1975
- Ziel: Möglichst viele Schwächen im Dokument finden
- Sehr aufwändiger Teamprozess:
  1. Planung: Team zusammenstellen (Autor, Moderator, Schriftführer, 2 bis 5 Gutachter)
  2. Einführungstreffen: Autor und Moderator erklären Produkt und Inspektionsziele
  3. Lesephase: Gutachter sehen Produkt durch und machen sich damit vertraut
  4. Inspektionstreffen: Team sucht unter der Leitung des Moderators nach Mängeln, Schriftführer protokolliert Ergebnisse, Autor fragt ggf. nach Klarstellung
  5. Autor korrigiert Produkt, Moderator kontrolliert
  6. Sammlung statistischer Daten (Größe, Aufwand, Defekte)
  7. Evtl. neue Inspektion des korrigierten Dokuments

Eine Lesetechnik: Perspektiven-basiertes Lesen

Freie Universität Berlin

Perspective-based Reading (PBR):

- Besonders geeignet für
  - natürlichsprachliche Dokumente in frühen Phasen (Anforderungen, Architektur, Grobentwurf, GUI-Entwurf)
  - Code
- Es gibt mehrere Gutachter
- Jeder verwendet zur Analyse eine andere Perspektive, z.B.:
  - Endbenutzer (evtl. verschiedene Benutzergruppen)
  - Entwerfer, Implementierer
  - Tester etc.
- Zu jeder Perspektive gehören andere Fragen und Schwerpunkte und andere Arten von möglichen Mängeln
  - Jeder Gutachter bekommt eine Beschreibung, die seine Perspektive erklärt (inkl. Checklisten)
- Erhöht Effektivität durch Senken der Überlappung

Andere Lesetechniken

- Object-Oriented Reading Techniques (OORTs)
  - Familie von Lesetechniken für Architektur- und Grobentwürfe
- Use-Based Reading (UBR)
  - Familie von Lesetechniken für Entwürfe von Benutzungsschnittstellen
- Defect-Based Reading (DBR)
  - Spezialverfahren für Anforderungsdokumente in Form von Zustandsautomaten

Beispiel 3: Selbst-Begutachtung (sehr informell)

Freie Universität Berlin

- Anderes Extrem: Der Autor sieht sein Produkt allein durch
- Kann sehr effektiv sein
  - Reales Beispiel:
    - 4 Monate Arbeit, dann 2 Tage Durchsicht,
    - 45% Defekt-Entdeckungsquote,
    - Rest durch 2 Monate Test
      - Hat also wohl weitere 1-2 Monate Testzeit eingespart!
  - Effektiv vor allem, wenn typische persönliche Fehlerarten bekannt sind
  - Und wenn die Begutachtungsgeschwindigkeit empirisch optimiert wird
- Nutzen hängt stark von Einstellung ab
  - "Ich will viele Defekte aufdecken"
  - Ein paar Tage Abstand nach der Entwicklung sind empfehlenswert

Beispiel 4: Werkzeuggestützte Code Reviews

Freie Universität Berlin

Heute weitest verbreiteter Typ von Durchsichten

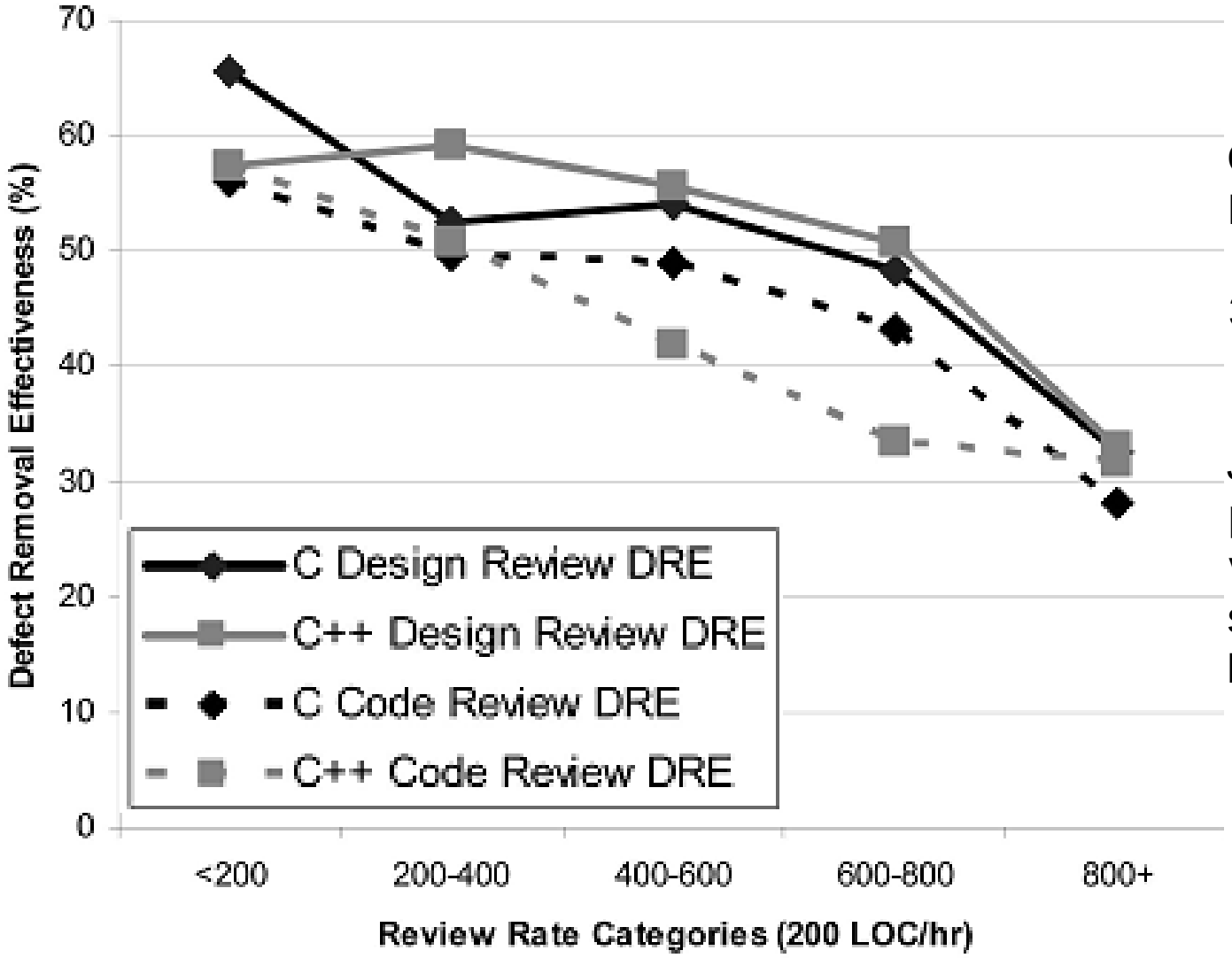
- Oft formalisiert und routinemäßig
  - über Werkzeuge wie GitHub ("pull requests") oder Gerrit
- Tendenziell feinkörning
  - Dutzende Zeilen, nicht Hunderte
  - Vorteil: Potentiell gründlich
  - Nachteil: wenig Überblick über ein größeres Ganzes??

- Auch anwendbar auf Dokumentation, Konfiguration, Testfälle, etc.

Empirische Resultate über Durchsichten

- Codedurchsichten finden mehr Defekte pro Stunde als Test
- Codedurchsichten finden andere Defekte als Test
- Wichtigste Steuergröße: Begutachtungsgeschwindigkeit
  - für Code: günstig sind ca. 50–300 Zeilen pro Stunde
- Fagan-Codeinspektionen sind ineffizient
  - große Überschneidung der Resultate verschiedener Gutachter
  - Kein Mehrwert durch Treffen: Separate Durchsicht reicht
  - Lange Laufzeit (Kalendertage): Prozessverzögerung
- Selbstbegutachtung kann sehr effektiv sein
  - Verlangt jedoch den Willen und die Geduld
- Durchsichten in frühen Phasen sparen besonders viel Kosten
- Perspektiven-basiertes Lesen:
  - Effektiver als wenn jeder alles sucht
  - Effizient durch hohe Gesamtausbeute

Begutachtungsgeschwindigkeit (Bsp.)

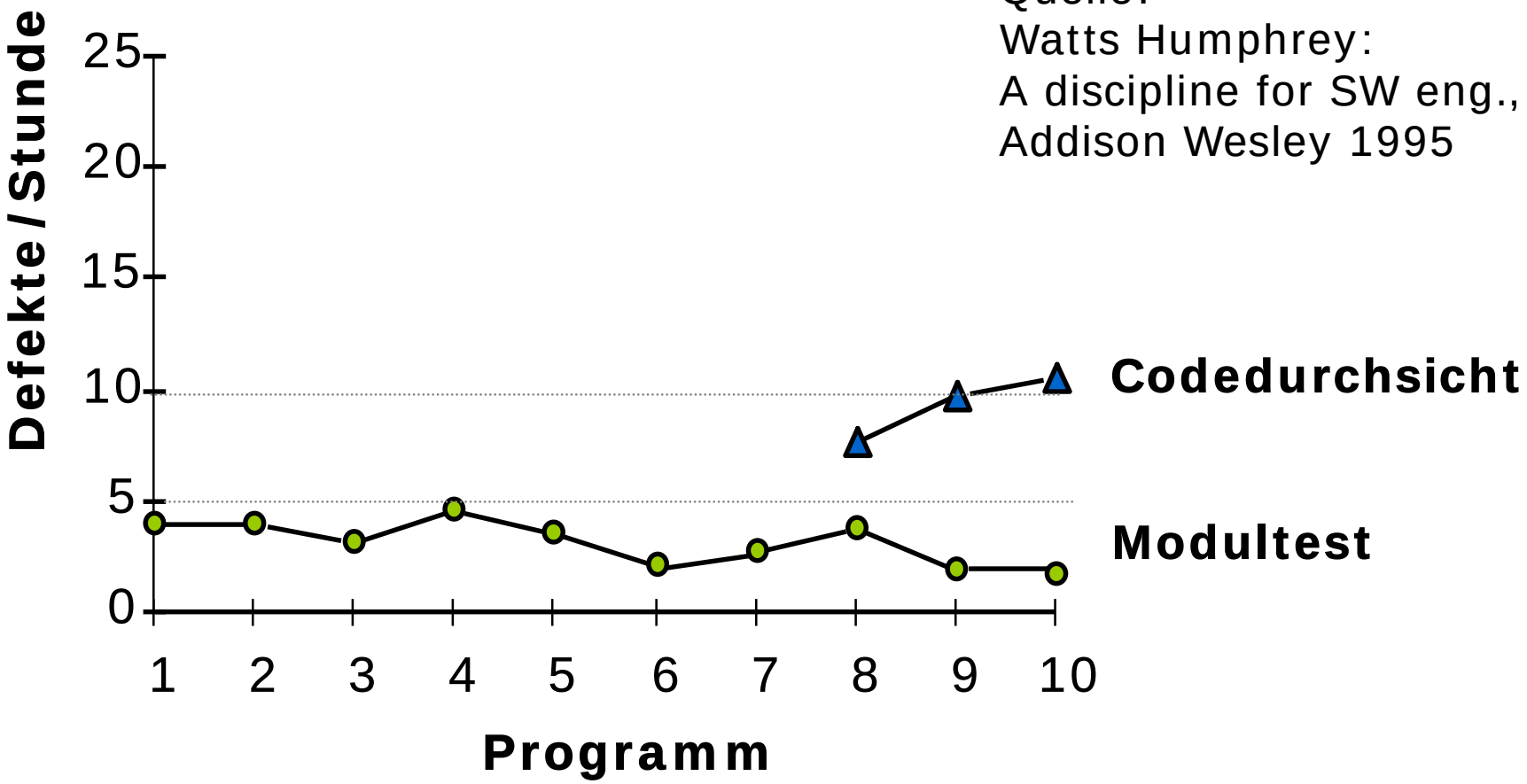


Quelle:
Kemerer, Paulk, IEEE Trans. on SE 35(4), July 2009

Jeder Punkt re-präsentiert Daten von >100 Durch-sichten (von vielen Programmierern)

- Beispiel: 20 Studenten, je 10 kleine selbstgeschr. Progr., erst Codedurchsicht (ab Programm 8), dann Test

Quelle:
Watts Humphrey:
A discipline for SW eng.,
Addison Wesley 1995



Übersicht

Analytische QS:

- Dynamische Verfahren (Test)
  - Defekttest
  - Benutzbarkeitstest
  - Lasttest
  - Akzeptanztest

- **Statische Verfahren**
  - Manuelle Verfahren
    - Durchsichten, Inspektionen
  - **Automatische Verfahren**
    - **Modellprüfung**
    - **Quelltextanalyse**

Konstruktive QS:

- Test- und Durchsichtsmgmt.
- Prozessmanagement
- Projektmanagement, Risikomanagement

Modellprüfung: Zustandsautomaten

- Für voll spezifizierte (Zustands)Automaten kann der gesamte Zustandsraum untersucht werden
  - Prüfung von Sicherheitseigenschaften, d.h. ob der Automat etwas tun kann, was er nicht tun soll
  - d.h. z.B. Beantwortung von Fragen der Art "Kann eine Abfolge A, B auftreten?"
    - Naives Bsp. für automatische Steuerung eines Containerkrans: "Kann ANHEBEN, LOSLASSEN geschehen?" (also ohne ABSENKEN dazwischen)

- Es gibt spezialisierte Programme, die selbst große Zustandsräume (10^{20} Zustände) effizient prüfen können
  - Forschungsgebiet "model checking"

Modellprüfung: Allg. Spezifikationsprüfung

- Auch andere Arten strukturierter Spezifikationen können auf bestimmte Eigenschaften automatisch geprüft werden:
  - Manche Arten von Inkonsistenz (innere Widersprüche)
  - Manche Arten von Unvollständigkeit
    - z.B. bei Fallunterscheidungen über Aufzählungstypen

```
switch(opera)
   case '+':

   case '-':

   case '*':

   case '/':

   default: p
```

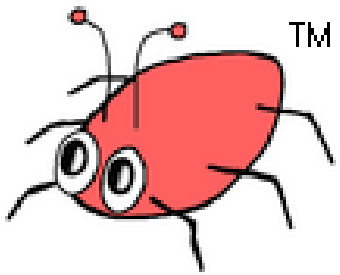
Wichtige Spezialfälle:

- Natürlichsprachliche Anforderungsbeschreibungen:
  - Vollständigkeit von Fallunterscheidungen oft überprüfbar
  - und bei entsprechender Dokumentstruktur automatisierbar

- Programmcode in typisierten Programmiersprachen:
  - Typprüfungen zur Übersetzungs- und Ladezeit sind eine Form der Konsistenzprüfung

Quelltextanalyse

- Für viele gängige Programmiersprachen gibt es Werkzeuge, die Quellcode auf diverse gängige Defekte oder Schwächen abklopfen
- z.B. für Java
  - dubiose *catch*-Konstrukte, fehlende *finally*-Klauseln, verdächtige Verwendungen von *wait()/notify()* etc.
  - siehe z.B. "FindBugs" http://findbugs.sf.net/
- z.B. für C/C++
  - Verletzungen der Typsicherheit
  - Verdächtige Verwendung von *malloc()/free()*
- etc.



- Große Unterschiede in der Leistungsfähigkeit der Werkzeuge
  - offenes Forschungsgebiet

Übersicht

Analytische QS:

- Dynamische Verfahren (Test)
  - Defekttest
  - Testautomatisierung
  - Benutzbarkeitstest
  - Lasttest
  - Akzeptanztest
- Statische Verfahren
  - Manuelle Verfahren
    - Durchsichten, Inspektionen
  - Automatische Verfahren
    - Modellprüfung
    - Quelltextanalyse

Sie befinden sich hier

Konstruktive QS:

- Test- und Durchsichtsmgmt.
- Prozessmanagement
- Projektmanagement, Risikomanagement

Danke!